# Scan Directed Load Balancing for Highly-Parallel Mesh-Connected Computers[1]

**Edoardo S. Biagioni**
**Jan F. Prins**
Department of Computer Science
University of North Carolina
Chapel Hill, N.C. 27599-3175 USA
biagioni@cs.unc.edu
prins@cs.unc.edu

**Abstract**

Scan Directed Load Balancing is a new, locality-preserving, dynamic load balancing algorithm for grid-based computations on mesh-connected parallel computers. Scans are used to efficiently determine what areas of the machine are heavily loaded and what areas are lightly loaded, and to organize the movement of data. Data is shifted along the mesh in a regular fashion to balance the load. The Locality Property of the algorithm guarantees that all the neighbors of a data point on the grid are stored either on the same processor, or on a processor that is directly connected to it.

Scan Directed Load Balancing is applicable to both SIMD and MIMD mesh-connected parallel computers, and has been implemented on the MasPar MP-1. We present some theoretical bounds achieved by the algorithm as well as the algorithm's performance on a particular image processing problem, edge-directed diffusion. Our experiments show that the algorithm is effective in improving the load distribution for real problems, while the efficiency of the original grid-based computation is preserved by the locality property.

[1] This work supported in part by Office of Naval Research contract #N00014-86-K-0680





To appear as book chapter in "Unstructured Scientific Computation on Scalable Multiprocessors," MIT Press, 1991.

## Introduction

A large class of scientific and engineering problems can be solved by repeated local computations at every point of data arranged in some regular grid. A classic example is the explicit solution of differential equations such as those describing the diffusion of heat through a surface. Other examples include the simulation of cellular automata and operations on image data. In each case the computation at a given grid element is *local* because it depends only on the element and its nearby neighbors on the grid.

A mesh-connected parallel computer is characterized by processors arranged in a regular grid with an interconnection network providing *local* communication between adjacent processors in the grid. Global communication between arbitrary processors is achieved by routing data over successive links in the network. Global communication delays scale with the number of processors whereas local communication delays are not subject to such a relationship. Consequently grid-based computations that are local in nature are ideally suited for execution on mesh-connected computers when the data grid can be put into correspondence with the mesh structure of the machine. Indeed the earliest demonstrations of success with scalable parallel computers were on such problems: heat diffusion on Illiac IV [13] and Ising spin computations on the DAP [11].

In this paper we are concerned with techniques to efficiently execute a class of such local computations in a setting where the data grid is large relative to the number of processors, and the local computations exhibit fixed points which, when attained, do not require further recomputation while data values in the local neighborhood remain unchanged.

Since, in a computation on a large grid, each processor is responsible for multiple data points, the presence of local fixed points makes possible a more efficient execution technique in which each processor only evaluates "active" points in the portion of the grid for which it is responsible. The problem arises, however, that the distribution of active points may become unpredictable as the computation evolves and may be highly irregular. Consequently, a static assignment of grid points to processors may lead to uneven numbers of active data points at each processor, reducing the efficiency of the parallel computation. In the worst case one processor holds all the active points, and no parallel speedup is attained

| Symbol | Value or Definition | Explanation |
|---|---|---|
| $k$ | $1\ldots$ | dimension of the mesh and data |
| $x$ | $0\ldots k-1$ | index over dimensions |
| $p^k$ | $p \gg 1$ | number of processors |
| $i_x$ | $0\ldots p-1$ | processor index along dimension $x$ |
| $I$ | $[i_0,\ldots,i_{k-1}]$ | processor index |
| $I_m$ | $[(p-1),\ldots,(p-1)]$ | maximum processor index |
| $p_I$ | $\ldots$ | processor at index $I$ |
| $p_x(i)$ | $\ldots$ | hyperplane $p_I$ where $I_x = i$ |
| $A_I$ | $\ldots$ | active points on processor $p_I$ |
| $A_{tot}$ | $\Sigma_I A_I$ | number of active points |
| $n^k$ | $\ldots$ | number of data points |
| $h_x$ | $0\ldots n-1$ | data index along dimension $x$ |
| $H$ | $[h_0,\ldots,h_{k-1}]$ | data point index |
| $D_H$ | $\ldots$ | data point at index $H$ |
| $\mu(H)$ | $\mu : H \rightarrow I$ | mapping of data points to processors |
| $b^k$ | $\ldots$ | buffer space per processor |
| $\eta_x(i)$ | $1\ldots$ | max data index on $x$-hyperplane $i$ |
| $g_x$ | $1\ldots\eta_x(i)$ | index over data points in processors $i$ |
| $d_x(i,g_x)$ | $\ldots$ | data hyperplane at index $g_x$ in $p_x(i)$ |
| $G$ | $[g_0,\ldots,g_{k-1}]$ | data point index within a processor |
| $\delta(U,V)$ | $\Sigma_i\lvert U_i - V_i\rvert$ | Manhattan distance between $U$ and $V$ |
| $\nu(V)$ | $\{V' : \delta(V,V') = 1\}$ | indices of neighbors of index $V$ |
| $max_x(A)$ | $MAX_{J\in(I-i_x)}A_J$ | $max$-reduction across axis $x$ of $A$ |
| $W_x(i,g)$ | $max_x(A)$ | activity of data $x$-hyperplane $(i,g)$ |
| $S_{x,i}$ | $\Sigma_{h:i=\mu_x(h)}W_{x,h}$ | activity of processor $x$-hyperplane $i$ |
| $\bar{A}$ | $max_x(A)/p$ | average load per processor hyperplane |
| $D$ | $n^k/p$ | points per processor per dimension |
| $\sigma_i(V)$ | $\Sigma_{j<i}V_j$ | left-to-right plus-scan of $V$ |
| $L_i$ | $\sigma_i(max_x(A))$ | activity over hyperplanes $< i$ |
| $F_i$ | $\bar{A} * i - L_i$ | active point flow, $i$ to $i-1$ |
| $\alpha$ | $\bar{A}/(b^k - D)$ | load equivalent of inactive data point |

Table 1
Notation

at all.

To maintain parallel efficiency, a dynamic load balancing strategy can be used to redistribute the data grid over processors periodically in response to the observed distribution of active points. Dynamic load balancing strategies have been studied extensively (e.g. [5],[3]) for large-grain computations in small-scale parallel systems where communication is not a limiting issue. In such a setting, migration of a computation away from the data it references is not considered to be a critical issue. In a scalable parallel system, however, this migration leads to the loss of the local communication property, reducing the scalability of the computation. This paper presents a dynamic load-balancing technique that *preserves* locality of communication in the computation.

The technique is applicable to mesh-connected MIMD or SIMD machines, including boolean hypercube machines, although the implementations reported on in this paper are for the MasPar MP-1, a two-dimensional mesh-connected SIMD machine. The load balancing computation uses global information to balance the load over the full machine; all global communication is in the form of scans (parallel prefix operations), which are efficiently implemented on scalable machines [2].

The rest of this paper is organized as follows. The next section describes our assumptions and gives some definitions and notation (in Table 1) to be used in subsequent sections. Section 3 describes Scan Directed Load Balancing in one-dimensional arrays of processors, and Section 4 extends Scan Directed Load Balancing to $k$ dimensions. Section 5 presents some experimental results from two-dimensional load balancing in an image processing problem. The last section reviews the characteristics of Scan Directed Load Balancing and points out directions for future research.

## Decomposition of Data into Processors

We assume the data $D$ is arranged in a $k$-dimensional grid in which every axis has length $n$ for a total of $N = n^k$ elements, and that the computation will be carried out on a $k$-dimensional mesh-connected machine $M$ with $P = p^k$ processors, with $p < n$ (for $p \geq n$ load balancing is trivial).

The dimensionality of the target machine mesh is the same as the dimensionality of the data grid for simplicity of exposition. Standard mem-

ory virtualization techniques can be used [6] to simulate a $k$-dimensional mesh efficiently in a lower-dimensional mesh, and grid embedding strategies exist [7] to simulate a $k$-dimensional mesh in a higher dimensional mesh (such as a boolean hypercube). A grey-code embedding insures that the constant cost for local communication in the $k$-dimensional mesh is preserved. These methods and the techniques to be presented are also readily generalized to handle a rectangular processor mesh or data grid.

A *domain decomposition* $\mu$ maps each element index $H = [h_0, ..., h_{k-1}]$ of the grid $D$ to a processor index $\mu(H)$ in $M$. The *hierarchical* decomposition $\mu(H)_i = \lfloor h_i \frac{p}{n} \rfloor$ partitions $D$ into $k$-dimensional subcubes of approximately equal volume (to be precise, each side of the subcube has length $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$). This decomposition minimizes communication in the evaluation of a local function applied at every point on the data grid.

The hierarchical decomposition is an instance of an *orthogonal* decomposition. A decomposition $\mu$ of $D$ is orthogonal if it can be written as $\mu([h_0, ..., h_{k-1}]) = [\mu_0(h_0), ..., \mu_{k-1}(h_{k-1}]$, where $\mu_0, ..., \mu_{k-1}$ are functions with domain $0..n-1$ and range $0..p-1$. *In other words*, an orthogonal decomposition maps each dimension of the data grid independently. If each of the functions $\mu_i$ is monotonically nondecreasing and surjective, then neighboring points on the data grid will always be found in the same or neighboring processors in the machine.

Scan Directed Load Balancing maintains an orthogonal decomposition $\mu$ that varies in response to the measured distribution of active cells.

We can restate these definitions more formally using the notation in Table 1. Each step of the computation we are load balancing is of the form $D_H^{t+1} = f(D_H^t, D_{\nu(H)}^t)$ for some function $f$, where $H$ ranges over the data point indices, and $\nu(H)$ is the set of indices which form the neighborhood of $H$. At all times we will want the decomposition $\mu$ of $D$ to respect the following property.

**Locality Property**: The processor distance between $D_H$ and each of the $2^k$ points in $D_{\nu(H)}$ is either 0 or 1.

We assume that the grid is initially decomposed hierarchically into the machine.

LEMMA 1 Given $n \geq p$, the hierarchical decomposition in which index $H = [h_1..h_k]$ is assigned to processor $p_I$, where $I = [i_1 \ldots i_k]$ and $i_x = \lfloor h_i \frac{p}{n} \rfloor$, has the Locality Property.

*Proof* The processor distance between the data point with index $H$ and any one of its neighbors with index $H'$ is $\delta(\mu(H), \mu(H')) = \delta(I, I')$. From the definition of neighbors (Table 1) we know that $\delta(H, H') = 1$, so $\exists x : (|h_x - h'_x| = 1) \wedge (\forall y \neq x, (h_y = h'_y))$. From the definition of $i_x$ and from $h_y = h'_y$, it follows that $i_y = \lfloor h_y \frac{p}{n} \rfloor = \lfloor h'_y \frac{p}{n} \rfloor = i'_y$. Therefore, $\delta(I, I') = |i_x - i'_x| = |\lfloor h_x \frac{p}{n} \rfloor - \lfloor h'_x \frac{p}{n} \rfloor|$. Since $n \geq p$, either $\delta(I, I') = 0$ or $\delta(I, I') = 1$, and the locality property holds. □

The following conditions on $\mu$ are preserved by the load balancing algorithm, and are sufficient to ensure that the decomposition has the locality property.

---

For any data index $H = [h_0, ..., h_{k-1}]$ and any axis $0 \leq x < k$

(1) $\quad h_x < h'_x \Rightarrow \mu(H)_x \leq \mu(H')_x$

(2) $\quad h_x = h'_x \Rightarrow \mu(H)_x = \mu(H')_x$

and for any processor index $I$

(3) $\quad \exists H : \mu(H) = I$

**Figure 1**
Locality Constraints

---

LEMMA 2 If the locality constraints hold, $\mu(H)_x < \mu(H')_x \Rightarrow h_x < h'_x$.

*Proof* Given $\mu(H)_x < \mu(H')_x$, $h'_x < h_x$ violates constraint 1, and $h_x = h'_x$ violates constraint 2; therefore, $\mu(H)_x < \mu(H')_x \Rightarrow h_x < h'_x$. □

THEOREM 1 If the Locality Constraints are satisfied, the Locality Property holds.

*Proof* If Theorem 1 didn't hold, it would be possible to destroy the locality property without violating any of the constraints. Then there

would be at least two data points, $D_H$ on processor $I = \mu(H)$ and its neighbor $D_{H'}$ on processor $I' = \mu(H')$, such that (a) $\delta(H, H') = 1 (by definition of \nu)$ and (b) $\delta(I, I') > 1$. Then either (1) $I$ and $I'$ differ only in dimension $x$ so that $\forall y \neq x, (i_y = i'_y)$, or (2) $I$ and $I'$ differ in at least two dimensions $x$ and $y$.

If (1), assume without loss of generality that $i_x < i'_x$. By (b), $i'_x - i_x > 1$, so there is some processor $J$, $i_x < j_x < i'_x$, that by Property 3 holds at least one data point $D_{H''}$, $H''$ such that $J = \mu(H'')$. Then, by Lemma 2, $h_x < h''_x < h'_x$, and $\delta(H, H') > 1$, which contradicts (a).

Conversely if (2) holds, $I$ and $I'$ are different in at least two dimensions $x$ and $y$. By definition of $\delta$, since $\delta(H, H') = 1$, $H$ and $H'$ differ in at most one dimension, $z$, so either $x \neq z$ or $y \neq z$ or both. If $x \neq z$ then $h_x = h'_x$ violates Constraint 2 since $\mu(H)_x \neq \mu(H')_x$; likewise if $y \neq z$.

Therefore, the Locality Property is preserved by any algorithm that respects the Locality Constraints. □

## One-Dimensional Scan Directed Load Balancing

This section describes Scan Directed Load Balancing for cases in which $k = 1$, so the computer is a linear array of processors and the data is a linear array of data points.

The code for one-dimensional Scan Directed Load Balancing has two steps, shown in Algorithm 1. The first step does storage **computation**, by computing the difference between the old mapping $I = \mu(H)$ and the new mapping $I' = \mu'(H)$. More exactly, what is computed is the number of active points that must be transferred between every pair of contiguous processors; we call this quantity the **flow** of active points between processors. A scan gives the load to the left of each processor; the difference between this and the desired load gives $F_i$, the flow of active points between processors. The second step does data **movement**, shifting the data from one processor to another until each data point $D_H$ is in processor $p_{\mu(H)}$. The function `put()` puts the leftmost data point of each active $p_i$ onto the rightmost end of the data buffer of $p_{i-1}$; the function `get()` takes the rightmost data point from $p_{i-1}$ and stores it as the leftmost point in $p_i$.

In the computation step, the number of active data points that must

```
1. Compute flows
   L_i ← σ_i(A)                                  activity to left of processor i
   Ā ← (L_{p-1} + A_{p-1})/p                     Ā broadcast to all processors
   if (Ā < 1) then abort load balancing endif    avoid underflows
   F_i ← iĀ − L_i                                flow from processor i to i − 1
2. Shift data
   forall (i) in parallel
     while (F_i > 1/2)                           tested by each processor
       if active (D_0) then F_i ← F_i − 1 endif
       put ()
       shift-left D_i                            shift own data buffer left by 1
     endwhile
     while (F_i < −1/2)
       shift-right D_i                           shift own data buffer right by 1
       get ()
       if active (D_0) then F_i ← F_i + 1 endif
     endwhile
   endfor
```

**Figure 2**
Algorithm 1. One-dimensional Scan Directed Load Balancing

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Proc. index i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Activity A | 2 | 0 | 0 | 1 | 0 | 1 | 2 | 2 |
| Left-activity L | 0 | 2 | 2 | 2 | 3 | 3 | 4 | 6 |
| Optimal load $\bar{A}$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Act pt flow F | 0 | -1 | 0 | 1 | 2 | 2 | 2 | 1 |

**Figure 3**
Example of 1-dimensional load balancing on a linear array of 8 processors, showing initial distribution (top) and final distribution (bottom)

flow through the left boundary of each processor is computed from the processor index, the average number of active data points per processor, and the number of active data points to the left of the processor. The broadcast operation by which $\bar{A}$ is communicated to all processors can be implemented by a right-to-left copy scan. The computation step checks that the average load is at least one; if it is less than one, any adjacent active data points would be distributed over more than two processors, violating the third locality constraint, so the load balancing is aborted. The computation step for a simple example is shown in detail in Figure 3, together with the results of the storage movement step.

The storage movement step has two phases. First, processors with $F_i > 0$ shift points to the left until $F_i$ active points have been shifted. Then, processors with $F_i < 0$ take points from the left until $-F_i$ active data points have been shifted. Since $F_0$ and $F_p$ are both zero, after the shift there are $L_i + F_i = i\bar{A}$ (actually, $\lfloor i\bar{A} \rfloor \leq L_i + F_i \leq \lceil i\bar{A} \rceil$) active points to the left of each processor $i$, at which point the load is balanced. If only active points are shifted, no processor will ever have to hold more active points than it already has or $\lceil \bar{A} \rceil$, whichever is greater, as long as all processors with $F_i > 1/2$ shift on every cycle in the first phase of data movement, and all processors with $-F_i < -1/2$ do likewise in the second phase.

Although Algorithm 1 achieves perfect load balance to within integer constraints, there is no limitation on the number of inactive data points that may end up in a single processor. For example, if all the active data is in one half of the machine and an equal amount of inactive data is in the other half of the machine, an even load distribution will push all the inactive data into the single processor at the inactive end, which can be a problem since the per-processor memory of highly parallel computers is often limited.

Some programs have an uneven distribution of data points but an even load per data point, and these programs will balance storage by balancing the load. Other programs have an even distribution of points but an uneven distribution of load per data point, and for these, prevention of buffer overflows is important. Algorithm 2 avoids buffer overflows by giving a load value to inactive data points; this insures that no processor will have more than $b$ total points, where $b$ is the maximum number of points that a processor can store.

Algorithm 2 is similar to Algorithm 1. The computation phase first

```
1. Compute flows
   L_i ← σ_i(A)                                  activity to left of processor i
   Ā ← (L_{p-1} + A_{p-1})/p                     Ā computed on processor p − 1
   if (Ā < 1) then abort load balancing endif    avoid underflows
   α ← Ā/(b − D̄)                                 p − 1 computes, broadcasts α
   A'_i ← A_i + αD_i                             compute modified load
   L'_i ← σ_i(A')                                new load to left of processor i
   Ā' ← L'_{p-1}/p                               Ā' = Āb/(b − D̄)
   F_i ← iĀ' − L'_i                              flow from processor i to i − 1
2. Shift data
   forall (i) in parallel
     while (F_i > (1 + α)/2)                     tested by each processor
       if active (D_0) then F_i ← F_i − (1 + α)
       else F_i ← F_i − α endif
       put ()
       shift-left D_i                            shift own data buffer left by 1
     endwhile
     while (F_i < −(1 + α)/2)
       shift-right D_i                           shift own data buffer right by 1
       get ()
       if active (D_0) then F_i ← F_i + (1 + α)
       else F_i ← F_i + α endif
     endwhile
   endfor
```

**Figure 4**
Algorithm 2. One-dimensional Load Balancing with Limited Buffers

computes $\alpha$, the pseudo-load value for inactive processes. The value of $\alpha$ is selected to be such that if any processor has $b$ inactive processes, its total pseudo-load is exactly $\bar{A}'$, or, conversely, any processor with pseudo-load $\bar{A}'$ and no active processes has exactly $b$ data points. Since any processor with pseudo-load $\bar{A}'$ and any active points has fewer than $b$ points, this guarantees that no processor will ever need to store more than $b$ points.

The only difference between the storage movement phases of the two algorithms is the amount by which the flows, $F_i$, are updated when shifting out an active or an inactive data point.

Algorithm 1 computes a redistribution that gives a perfectly balanced load (within integer constraints) of

$$\lfloor \bar{A} \rfloor \leq A_i \leq \lceil \bar{A} \rceil$$

since $\bar{A} - 0.5 < A_i < \bar{A} + 0.5$ and $A$ is an integer.

Algorithm 2, on the other hand, only guarantees that

$$\bar{A}' - \frac{1+\alpha}{2} \leq A'_i \leq \bar{A}' + \frac{1+\alpha}{2}$$

where $\bar{A}' = \bar{A} + \alpha \bar{D}$. If each active point has a load of 1, the pseudo-load of an active point is $1 + \alpha$ and $A_i \leq A'_i/(1+\alpha)$. Since $\alpha = \bar{A}/(b - \bar{D})$, if $b \gg \bar{D}$, $\alpha$ is small and the performance of the two algorithms is comparable; if $\bar{D}$ is close to $b$, the maximum load on any processor may be noticeably higher than the average load.

The exact bound for $A_i$ in terms of $\bar{A}$, $\alpha$, and $\bar{D}$ is

$$A_i \leq \frac{\bar{A} + \alpha \bar{D} + (1+\alpha)/2}{1+\alpha}$$

which can be reformulated to exclude the artificial factor $\alpha$ and give

$$A_i \leq \frac{\bar{A}b + (\bar{A} + b - \bar{D})/2}{\bar{A} + b - \bar{D}}$$

If most data points are active, $\bar{A} \approx \bar{D}$ and the bound reduces to $A_i \leq \bar{A} + 1/2$, which is the bound for Algorithm 1. The worst-case performance is obtained when $b \approx \bar{D}$, in which case $A_i \leq \bar{D}$, the best bound that can be given, simply means very little redistribution is taking place and a processor may happen to have all of its $\bar{D}$ data points active. This corresponds to intuition, since if little or no buffer space is available, load

balancing is unable to contribute much to performance because there is no flexibility in distributing data.

Finally, the next theorem shows that the above load balancing algorithms satisfy the Locality Constraints and therefore preserve the Locality Property.

LEMMA 3 The `put()` and `get()` operations preserve the first Locality Constraint.

*Proof* The definition of `put()` says that `put()` transfers the leftmost data point on one processor onto the rightmost end of the data buffer of the neighbor to its left; this preserves property 1, since each buffer in each processor acts like a FIFO queue and no data point can ever "pass" another data point. The same is true of `get()`, which proves the lemma. □

THEOREM 2 Algorithm 1 preserves the Locality Property.

*Proof* To prove that Algorithm 1 preserves the Locality Property, assume that the distribution to which Algorithm 1 is applied already satisfies the Locality Constraints. The first constraint is preserved by Algorithm 1 since the only data movement operations are performed by `put()` and `get()`, which by Lemma 3 preserve the locality constraints. The second constraint is trivially true in one dimension because there are no dimensions orthogonal to the axis along which the load is balanced. The third constraint is satisfied by avoiding underflow: after the load is balanced each processor has a load between $\lceil \bar{A} \rceil$, and $\lfloor \bar{A} \rfloor$; if $1 \leq \lfloor \bar{A} \rfloor$, each processor at the end of the balance will have at least one active data point, thus satisfying constraint 3. In the case that $\bar{A} < 1$, no load balancing is done and constraint 3 is preserved. □

LEMMA 4 Algorithm 2 preserves the Locality Property.

*Proof* Constraints 1 and 2 are satisfied as for Algorithm 1. Constraint 3 is satisfied because the load on each processor is at least $\bar{A}' - (1+\alpha)/2$; since $\bar{A}' = \bar{A} + \alpha$ and $\bar{A} \geq 1$, the load is at least $1 + \alpha - (1+\alpha)/2 = (1+\alpha)/2 > 0$. Since every processor has a nonzero pseudo-load, every processor has at least one data point. □

While interest in one-dimensional scan directed load balancing might seem academic, there are actually many problems that can use just such a scheme. Simpler variants of Algorithm 1 have been used for garbage collection and storage management on both the Connection Machine [2] and on the FFP Machine [9] [1]. In addition, Algorithms 1 and 2 can be used on any computer composed of a linear array of processors; any multi-dimensional data set can be mapped to such a computer by a suitable function.

## Scan Directed Load Balancing for Higher Dimensional Meshes

The algorithms for one-dimensional Scan Directed Load Balancing can be used with slight changes for meshes of dimension $k > 1$. In this case both the processor mesh and data grid have the same dimension $k$. Multidimensional Scan Directed Load Balancing simply applies the one-dimensional algorithm to each dimension of the grid independently. When balancing dimension $x$, for instance, the algorithm treats the $k$-dimensional data as a single linear array of $(k-1)$-dimensional hyperplanes and balances the load by shifting along $x$ so that each hyperplane of processors has the same load.

The algorithms use *local* addresses for the data points, so a point on processor $p_I$ is identified by the index $G$ within the processor, so that the pair $(I, G)$ uniquely identifies a data point. A hyperplane of processors is identified by the single index $i$ and the dimension $x$ to which it is perpendicular, whereas a hyperplane of data is identified by the two indices $(i, g)$ and the dimension $x$.

The algorithm first reduces the loads on the processors to a single vector of loads by computing the hyperplane loads $W_x(i, g)$ for each data hyperplane; $W_x(i, g)$ is the maximum number of active points with $x$-coordinate $(i, g)$ within any single processor. $W_x(i, g)$ can be computed using segmented max-scans. The vector $S_{x,i}$ gives the load of each $x$-hyperplane of processors.

Once a single load has been computed for each hyperplane of processors, each hyperplane is treated as a point in the one-dimensional load balancing algorithm, as can be seen by comparing Algorithm 3 to Algorithm 1. The main difference is that in the one-dimensional case each

```
forall (x ∈ {1...k})
1. Compute flows in dimension x
   forall (g ∈ {0...η_x(i)})                          loop over data indices
     W_{i,g} ← max_x(A_x(g))                  activity on data hyperplane d_x(i, g)
   endfor
   S_i ← Σ_g(W_{i,g})                                 activity on hyperplane p_x(i)
   L_i ← σ_i(S_i)                              activity over hyperplanes p_x(j), j < i
   Ã ← (L_{p-1} + S_{p-1})/p                          broadcast to all processors
   if (Ã < max(h_x)) then abort endif                          avoid underflows
   F_i ← iÃ − L_i                          flow from hyperplane p_x(i) to p_x(i − 1)
2. Shift data along dimension x
   forall (i) in parallel
     while (F_i > W_{i,0}/2)                                tested by each processor
       F_i ← F_i − W_{i,0}
       put ()          put hyperplane d_x(i, 0) to processors p_x(i − 1)
       shift-down D_i, W_i                    shift to fill hyperplane d_x(i, 0)
     endwhile
     W_p ← W_{i−1,η_x(i−1)}                        load of topmost data of p_x(i − 1)
     while (F_i < −W_p/2)                                   tested by each processor
       shift-up D_i, W_i                      shift tc clear hyperplane d_x(i, 0)
       get ()                 get hyperplane d_x(i − 1, W_p) into d_x(i, 0)
       F_i ← F_i + W_{i,0}
       W_p ← W_{i−1,η_x(i−1)}          p_x(i − 1) may or may not call get ()
     endwhile
   endfor
endfor
```

**Figure 5**
Algorithm 3. Scan Directed Load Balancing for more than one dimension

point has a load of either 0 or 1, whereas in the multi-dimensional case each hyperplane of data may have a load considerably greater than 1; the loop termination and underflow prevention conditions of Algorithm 3 reflect this difference. Note that the functions `get ()` and `put ()` now move hyperplanes of data (rather than data points) between processors.

Algorithm 4 is the multi-dimensional analogue of Algorithm 2, and a straightforward extension of Algorithm 3, and is omitted for brevity. Each processor is assumed to have a $k$-dimensional buffer for data of size $b^k$.

The underflow condition in Algorithms 3 and 4 compares the active load to the maximum number of data points in a cell over all data hyperplanes perpendicular to the axis being balanced. As long as the average load is greater than this number, no data hyperplane will ever need to be distributed to more than one hyperplane of processors, which preserves the third Locality Constraint.

Unlike the one-dimensional case, the improvements in load balancing that are given by the multi-dimensional algorithm cannot be as neatly bounded as in the one-dimensional case. The reason for this is that the algorithm evens out the row and column loads independently, and in the worst case this may not be effective in optimizing the load per processor. An obvious worst case is when the entire load is evenly distributed among the processors in one diagonal of the mesh: since no amount of orthogonal shifts is going to improve the load balance, Scan Directed Load Balancing computes flows of zero. Such worst cases are rare in practice, as can be shown by our experiments.

## Experimental Data

An example of a computation to which these techniques can be applied is the edge-directed diffusion operation in image processing [12]. Edge-directed diffusion operates on image data arranged as a grid of intensity values (pixels). At each iteration a pixel's value is set to be a weighted average of the values of its neighbors and itself, with neighbors whose value differs greatly (i.e. those defining an edge) having low weight. Whenever the averaging operation yields a value that is within some threshold $\epsilon$ of the current value, the value is not updated and the local averaging function has achieved a fixed point.

When applied to an image such as the one in Figure 6, most pixels are modified on early iterations. Thereafter, activity tends to cluster into regions surrounded by edges, spreading slowly between regions. The computation terminates after some predetermined length of time or when all pixels have reached a fixed point. The result on the sample image is shown in Figure 7. The algorithm smooths noise in areas of the image that don't have sharp edges, but does not blur the edges themselves, so the overall sharpness of the image is preserved. Edge-directed diffusion is used at the University of North Carolina [4] for removing noise from medical images such as those produced by CT scanners.

Scan Directed Load Balancing has been used to optimize edge-directed diffusion. The algorithm has been coded to run on the MasPar MP-1 [10], a parallel SIMD computer with 4096 processors laid out in a $64 \times 64$ 2-dimensional mesh. Each processor of the MP-1 has a 4-bit wide ALU and connections to 8 neighbors, 4 in the orthogonal and 4 in the diagonal directions.

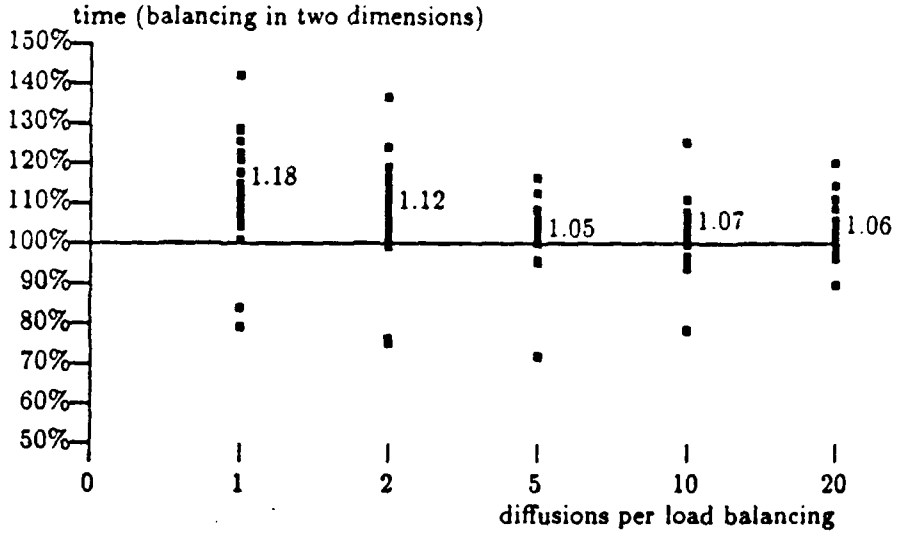
Even though the MP-1 provides a general-purpose router which is much faster for sending small amounts of data long distances, the relative speed and bandwidth of the mesh connection is much greater, so the diffusion program uses only the mesh for communication. The image is divided into tiles and the program performs the diffusion independently on each tile after copying the pixels from the edges of each of the 8 neighboring processors. The subdivision of the image into contiguous rectangular tiles minimizes the distance over which communication occurs and the total bandwidth of communication, so that significantly more time is spent computing the diffusion than communicating pixel values. Under these circumstances, load balancing can be useful in improving the performance of the program, as long as speedup given by load balancing is greater than the additional overhead due to load balancing.

The diffusion program was run on 32 medical and test image files of sizes $512 \times 512$ pixels (large), $400 \times 448$ pixels (medium), and $256 \times 256$ pixels (small). On the $64 \times 64$ processor MP-1 this gives per-processor image sizes of $8 \times 8$, $6 \times 8$, and $4 \times 4$ pixels respectively.

For each image, the performance of the diffusion algorithm alone was compared to the performance of the diffusion algorithm with load balancing. The two measures of performance we use are the maximum load on any given processor averaged over all diffusion cycles, and the

**Figure 6**
The original image

**Figure 7**
The image after running Edge Directed Diffusion



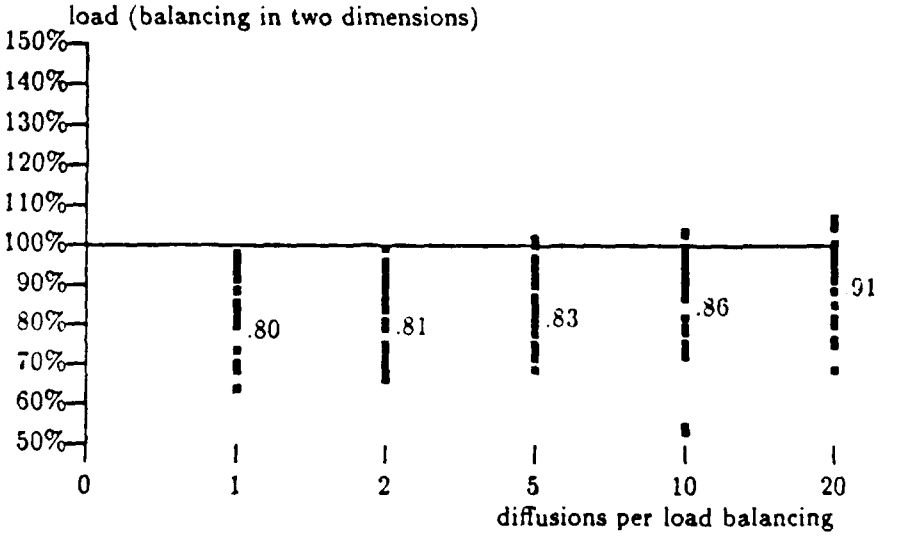




**Figure 8**
**Performance of Scan Directed Load Balancing**

actual time it took to complete the diffusion. The first measure is an indication of how successful the load balancing algorithm is in reducing the load of the processors. The second measure takes into account the costs of performing the load balancing and is a more accurate measure of the success of this implementation of Scan Directed Load Balancing in improving overall performance. Since different implementations might be more or less efficient than the present one, the first measure is more useful in assessing the algorithm's potential for improving performance, whereas the second measure gives the actual performance of the current implementation on the available hardware. Figures 8–10 give the actual measurements.

Each graph shows the performance of the load balancing algorithm for Scan Directed Load Balancing applied every iteration of the diffusion algorithm, every other iteration, every fifth iteration, and so on. The vertical axis gives the ratio of the cost using load balancing to the cost without load balancing.

The first observation is that there is a very wide spectrum of performances, from abysmal to very good. It was mentioned at the end of the previous section that performance cannot be predicted for the multi-dimensional case; an additional problem is that present load is not always an accurate predictor of future load, and so the algorithm occasionally makes the load worse than it was.

On average, the algorithm is quite effective in reducing the load. More important, the average load decreases monotonically as the algorithm is applied more and more frequently, which indicates that the algorithm reliably improves the load balance.

The average time is a shallow V-shaped curve, since more frequent applications of the algorithm mean not just better performance on the diffusion task but also higher overhead for the load balancing. For the present implementation, the optimal ratio of diffusion to load balancing cycles is in the neighborhood of five. The next section discusses additional hardware that could be provided to make it advantageous to run load balancing on every diffusion cycle and to reap all the potential benefits of load balancing.

Another interesting comparison is between images of different sizes. A comparison of the graphs for the performance on the small and large images shows that both the time and the load are better on the large images. The better load performance on large images can be explained

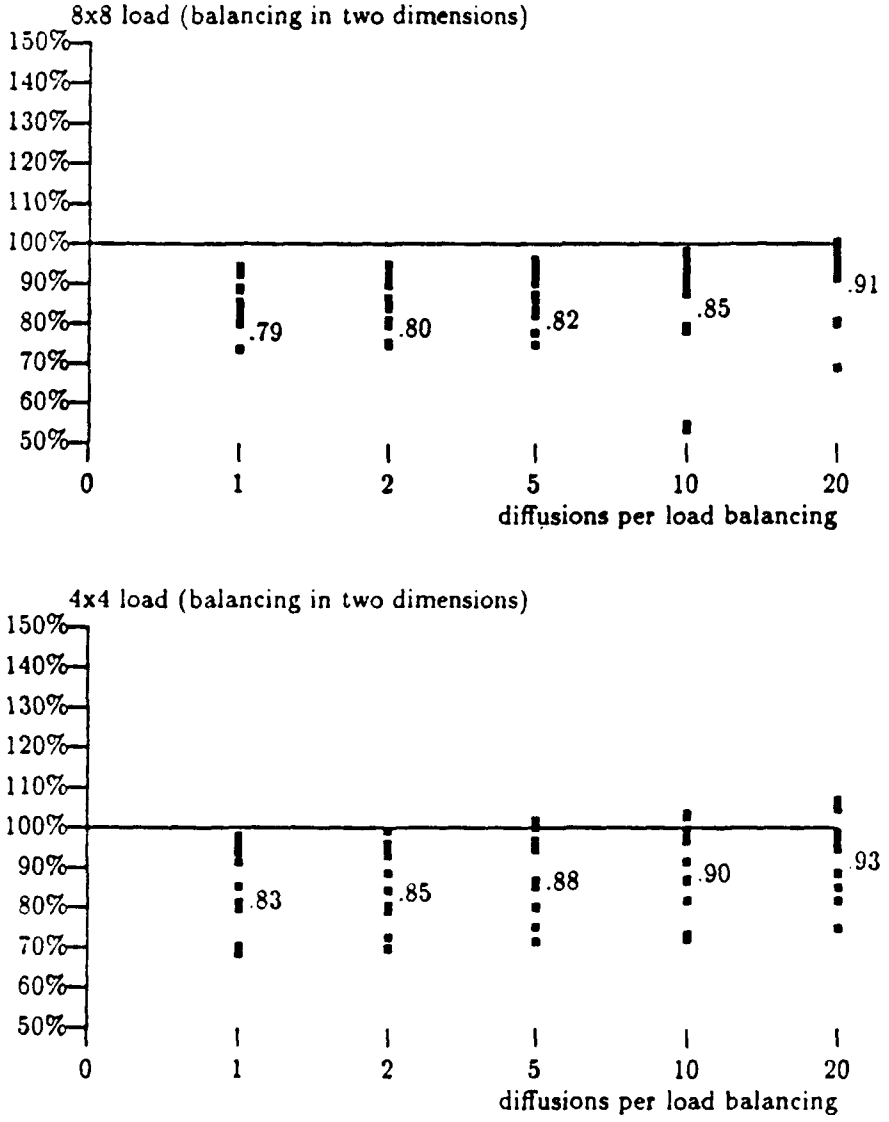
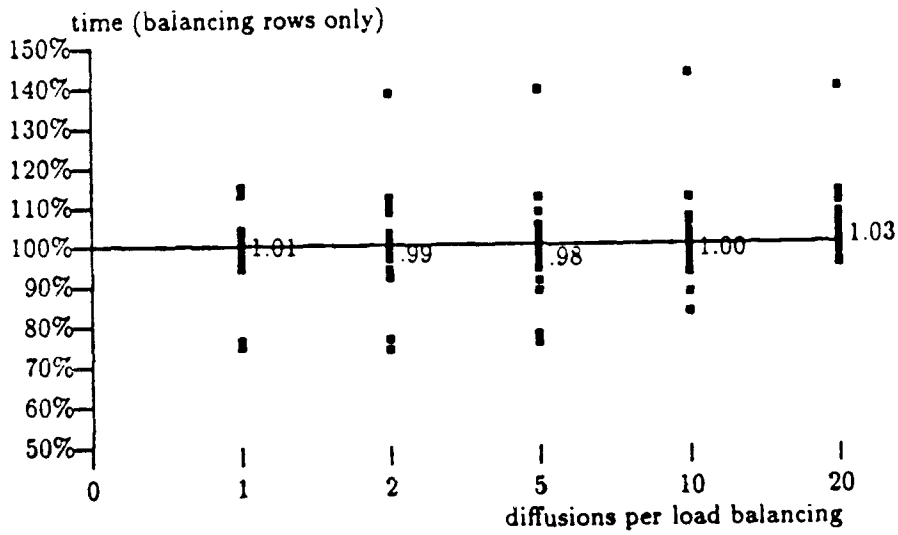



**Figure 9**
**Performance of Scan Directed Load Balancing for Different Sized Images**



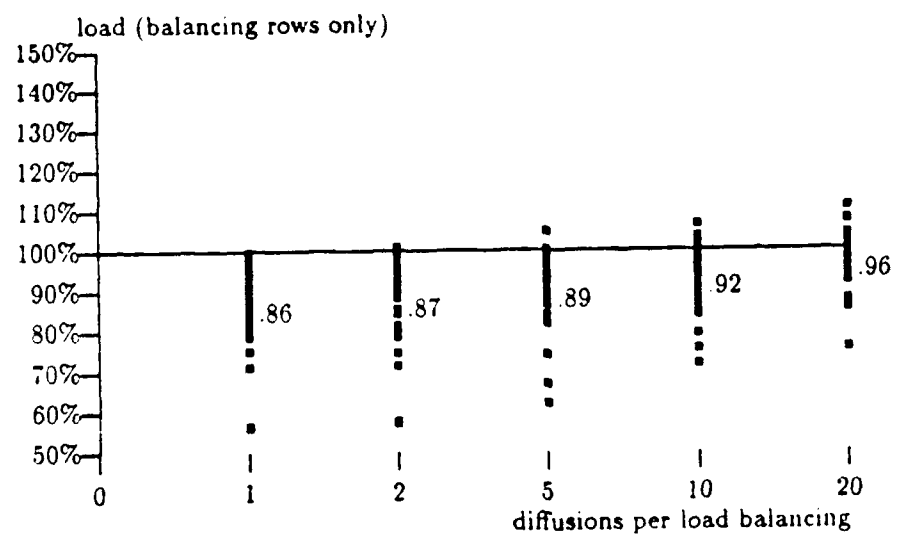




**Figure 10**
**Performance of Scan Directed Load Balancing in only one dimension**

by considering that the granularity of load balancing is finer on the larger images and the load balancing algorithm therefore can allocate the load more accurately. The better time performance on larger images is due to both the finer granularity and the fact that each diffusion cycle takes longer, so the overhead of load balancing is less significant than on the smaller images.

The better time performance on the one-dimensional load balancing is due in part to less overhead for the load balancing itself, and in part to reduction in the overhead of loop processing. The MP-1 provides local indirect addressing, which is used to build a worklist, but worklist processing is a noticeable fraction of the processing time, and applying the load balancing algorithm in only one dimension brings many of the benefits of Scan Directed Load Balancing while reducing many of the costs of two-dimensional load balancing. The average load is higher in the one-dimensional case, however, as might be expected from the fact that not all the benefits of load balancing can be realized.

## Summary and Concluding Remarks

The organized synchronous character of Scan Directed Load Balancing is a quite different from the distributed, demand-driven nature of many popular load balancing algorithms [8] [3]. Scan Directed Load Balancing does not require any kind of tuning and provides predictable performance for any range of loads. This predictable performance is particularly useful for parallel computers with large numbers of nodes and small-grain processes, in which an algorithm that isn't centrally organized may not be as quick at evening out load imbalances.

Scan Directed Load Balancing uses barrier synchronization and synchronous transfer of per-process data to guarantee that no shift will ever overflow a data buffer, and pseudo-loads for inactive processes if such processes are present. The pseudo-load mechanism trades off processor activity for processor space to simultaneously bound both load and buffer space. This algorithm does not always achieve perfect balance; interesting open questions are whether the optimal distribution that respects buffer size constraints can be efficiently computed in parallel, and whether the optimal distribution has better worst-case performance than Algorithm 2.

Scan Directed Load Balancing preserves the Locality Property, which simplifies programming of many problems and makes it easier to add load balancing to existing programs. The Locality Constraints guarantee nearest-neighbor access of neighboring data points, which is very well suited to SIMD-style conflict-free access of data over a mesh for programs that do mostly Local Communication. Guaranteeing the Locality Property by the Locality Constraints in multi-dimensional meshes can lead to load imbalances that can't be corrected, independently of the algorithm used to balance the load. Careful study of the Locality Constraints to reduce this problem may yield new algorithms that might be less generally applicable (since the SIMD style of mesh usage would have to be abandoned) but have better worst-case performance.

The measured performance of Scan Directed Load Balancing shows that such worst cases are rare in practice, and that the algorithm as it stands can effectively reduce the load of an average computation. The measured performance also shows that the present implementation is too slow to produce significant improvements for the anisotropic diffusion problem; this could be rectified either by improving the software implementation or by providing specialized hardware support for scans and shifts. The MasPar MP-1 currently uses the mesh for both scans and shifts; since these mechanisms are crucial to the performance of Scan Directed Load Balancing (as well as other algorithms), quantifying the advantage of hardware-assisted scans or shifts is a very important next step in our work. Also important is to test the Load Balancing algorithm on other applications so the performance of the algorithm itself can be evaluated independently of the specific problem. In general, the performance of Scan Directed Load Balancing should improve as the grain of the processes increases, and since the processor grain in edge directed diffusion is quite small (a diffusion cycle for one pixel is only a few operations), it is reasonable to expect that the algorithm will be effective for a wide variety of computations.

Scan Directed Load Balancing is a minimalist load balancing algorithm. It is well-suited for both SIMD and MIMD, is effective for both linear arrays and multi-dimensional meshes, only uses scans for computation, broadcasting and synchronization, and can be used to load balance any mesh-based program that is computation intensive. As a result, Scan Based Load Balancing is useful over a wide range of machine architectures and computations, and especially appropriate for

fine-grained computations on massively parallel architectures.

## Acknowledgements

We appreciate the input and suggestions made by V. Chi, G. Mago, and J. Welch, and the support of the people at MasPar Computer Corporation.

# Bibliography


[1] Edoardo S. Biagioni. *Scan Directed Load Balancing*. PhD thesis, University of North Carolina, Chapel Hill, 1991. in preparation.

[2] Guy E. Blelloch. The scans as primitive parallel operations. *IEEE Transaction on Computers*, 38(11), November 1989.

[3] Shyamal Chowdhury. The greedy load sharing algorithm. *Journal of Parallel and Distributed Computing*, 9(1), June 1990.

[4] Robert Cromartie and Stephen M. Pizer. Edge-affected context for adaptive contrast enhancement. In *12th International conference on Information Processing in Medical Imaging (IPMI '91)*, July 1991.

[5] John Zahorjan Derek L. Eager, Edward D. Lazowska. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, SE-12(5):662–675, May 1986.

[6] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lyzenga, Steve W. Otto, John K. Salmon, and David W. Walker. *Solving Problems on Concurrent Processors. volume I: General Techniques and Regular Problems*. Prentice-Hall, Englewood Cliffs, NY, 1988.

[7] Ching-Tien Ho and S. Lennart Johnsson. Embedding meshes in boolean cubes by graph decomposition. *Journal of Parallel and Distributed Computation*, 8(4):325–339, 1990.

[8] Frank C. H. Lin and Robert M. Keller. The gradient model load balancing method. *IEEE Transactions on Software Engineering*, SE-13(1):32–38, January 1987.

[9] Gyula A. Magó and Donald F. Stanat. The FFP Machine. In Veljko M. Milutinović, editor, *High-Level Language Computer Architectures*, pages 430–468. Computer Science Press, 1989.

[10] MasPar Computer Corporation, Sunnyvale, California. *MasPar Parallel Application Language (MPL) Reference Manual*, 1990.

[11] G. S. Pawley, K. C. Bowler, R. D. Kenway, and D. J. Wallace. Concurrency and parallelism in mc and md simulations in physics. *Comput. Phys. Commun.*, 37:251–260, 1985.

[12] P. Perona and J. Malik. Scale-space and edge detection using anisotropic diffusion. Technical Report UCB/CSD 88/483, Computer Science Division, University of California, Berkeley, California, December 1988.

[13] D. L. Slotnick. The fastest computer. *Scientific American*, 224(2):76–87, February 1971.